ParaNut - An Open, Scalable, and Highly Parallel Processor Architecture for FPGA-based Systems

Gundolf Kiefer*, Michael Seider†, and Michael Schaeferling*

*University of Applied Sciences Augsburg
Dept. of Computer Science
An der Hochschule 1
86161 Augsburg, Germany
Email: {gundolf.kiefer, michael.schaeferling }@hs-augsburg.de

†Mixed Mode GmbH
Lochhamer Schlag 17
82166 Graefelfing, Germany
Email: michael.seider@mixed-mode.de

***Abstract*—The paper presents the *ParaNut* architecture, a new open and highly scalable processor architecture for FPGA-based systems. The *ParaNut* architecture follows a new concept of parallel processing units with customizable capabilities which allows a seamless transition between SIMD vectorization (data-level parallelism) and simultaneous multi-threading (thread-level parallelism). A preliminary implementation of a *ParaNut* on a Virtex-5 FPGA achieves 6.13 CoreMarks/MHz using 8 cores, which is approximately 5 times faster than OpenRISC 1200 (1.28 CoreMarks/MHz).**

Keywords: Computer Architecture, RISC, Customizable Processor, Soft-Core, FPGA

I. Introduction

The capacity and efficiency of available field-programmable gate arrays (FPGAs) is growing continuously. Today, even low- or mid-range devices allow the implementation of complete Systems-on-a-Chip (SoC) or even Networks-on-a-Chip (NoC) on a single FPGA device. Traditionally, the strengths of FPGAs in embedded systems lie in the implementation of specialized hardware. However, complex SoCs virtually always require one or several CPUs to implement the software part of the system.

A growingly important application field for high-performance FPGAs is real-time image processing, for example, in Computer Vision applications [1]. Typical Computer Vision applications can be split into processing stages, classified into 4 classes, namely a) pixel-based and b) window-based filters as well as c) semi-global and d) global operations [2], [3]. Local and window-based filters are rather simple operations performed on each image pixel and as such are well-suited for SIMD (single instruction, multiple data) vectorization or specialized hardware. Semi-global and global operations are more complex and typically well-suited for parallelization on multiple CPU cores, which may be highly customized and application-tailored processors as in the case of the SURF ("Speeded-Up Robust Features") descriptor calculation presented in [2].

Also, for low-cost and low-performance products, a single FPGA with a (small) soft-core processor has some clear advantages over standard microcontrollers accompanied by an FPGA or CPLD for custom logic. The board design is simpler and more immune against chip discontinuations. If the system integrator owns the HDL code of the soft-core processor, he can eventually port the processor to a new FPGA type, and he does not need to change anything in the complex software.

Looking at the available soft-core processors for FPGAs today, it can be observed that they are either device-specific (e. g. Xilinx MicroBlaze, Altera Nios), not multi-core capable (e. g. OpenRISC 1000, LEON3 only up to 4 cores [4]), or too complex for small, area-limited systems (e. g. OpenSPARC). To address a large field of applications, a processor architecture should support a speed-area trade-off in a wide range. On the one hand, an area-optimized, pipeline-less single-core implementation should be possible for low-cost applications. On the other hand, for high-performance applications, a highly parallel many-core variant (8 cores or more) is desired that provides a shared memory model without noticeable performance degradation due to bus contention.

This paper presents the *ParaNut* architecture, a new open and highly scalable CPU architecture for FPGA-based systems. Special focus is put on parallelism at thread and data level in order to allow both small and power-efficient systems based on one architecture. The *ParaNut* architecture introduces a new concept of parallel processing units with customizable capabilities which allows a seamless transition between SIMD vectorization (data-level parallelism) and simultaneous multi-threading (thread-level parallelism). By sticking to an existing instruction set architecture (OpenRISC), a GCC-based tool chain for software development is already available today.

Section II gives an overview on presently available soft-core processors for FPGAs as well as computing platforms for embedded systems in general. Section III introduces the basic concepts and design decisions of the *ParaNut* architecture. The new programming concept that incorporates SIMD and multi-thread programming is sketched in Section IV. Section V describes the implementation status of the project, and is followed by experimental results in Section VI. Section VII concludes the paper.

II. Related Work

There are several approaches on how to integrate CPUs into FPGA-based platforms, namely by means of hard-core or

soft-core processors. Recent high-grade FPGAs include one or more dedicated hard-wired (hard-core) CPU cores, as for example Xilinx equips their Zynq-FPGAs with a dual-core ARM CPU. Although this concept offers a convenient platform for general applications, these FPGA devices are rather cost-intensive and the involved CPU architectures are largely fixed, which makes it hard to adapt them towards application-specific needs. A present trend in the area of Embedded System is to put a large number of CPU cores on a single chip to provide computational power. The Epiphany accelerator chip is a commercially available many-core solution, contains 16 or 64 CPU cores and can be coupled with an FPGA [5]. Although the development tools and libraries are open, the hardware itself is not, which prevents the system designer from customizing the CPU architecture to fit the application.

Soft-core processors can easily be adapted and optimized for a particular application. The idea to use an *architecture description language (ADL)* to define application-specific processors already evolved several years ago [6]. An ADL allows to synthesize the hardware together with compilers and other development tools automatically [7]. Projects following this approach are *MIMOLA* (University Dortmund), *LISA* (RWTH Aachen) and the commercial *Xtensa* project. Besides these projects, there is a number of research projects dealing with configurable multi-core and parallel architectures in general. Some of them are *CHIMERA* [8], *PipeRench* [9], *MORA* [10] and *LegUp* [11].

For productive use, quite a number of soft-core processors targeting FPGAs are available today. Typically, they can be configured to a certain extent, depending on the specific implementation which allows to customize their capabilities, in order to better fit the actual applications demands. Commercial solutions for soft-cores are, for example, the *MicroBlaze* (Xilinx) or *Nios II* (Altera) processors. Both of them are closed-source and only are applicable on the vendors FPGAs. Open-source projects allow to shape the architecture towards the specific application even more due to the available source code at register-transfer level (RTL) and are less restricted towards specific FPGA vendors. Examples for open-source processors are *LEON 2/3/4* (Gaisler Aeroflex), *OpenSPARC* (Sun) or *OpenRISC*. Most of the mentioned soft-core processors share one or several of the following drawbacks or limitations: They are either (a) not to be used freely (bound to license fees), (b) platform-specific, (c) not multi-core-capable or (d) have a high logic resource usage even in their smallest configuration.

III. The Basic Architecture

A. Design Considerations

Numerous techniques for exploiting parallelism in general-purpose processors are known today. They can be grouped into three categories, namely *data-level parallelism (DLP)*, *instruction-level parallelism (ILP)*, and *thread-level parallelism (TLP)* [12].

The DLP category includes SIMD (single instruction, multiple data) extensions, such as Intel's SSE or the ARM SIMD media extensions. From a hardware perspective, SIMD is very attractive since it can considerably reduce the load on the memory bus (i. e. the number of instruction fetches per data items processed). Unfortunately, the SIMD extensions of popular embedded or workstation processors share some properties that make it difficult to develop optimizing compilers utilizing such units automatically [12]. Hence, in practice, these SIMD extensions are utilized manually, either by using assembly programming or by using special libraries. Both ways are time consuming and lead to unportable code. Better support for high-level language programming is desirable from a software developer's perspective.

The ILP category includes all the techniques that improve the throughput of a single general-purpose instruction stream, such as (deep) pipelining, out-of-order execution, or VLIW (very long instruction word) architectures. ILP techniques generally come at the cost of high area usage and power consumption. They require additional hardware structures to resolve pipeline conflicts and to perform the instruction scheduling. On the other hand, machine instructions which depend on each other cannot be parallelized in many cases. Hence, the costs in terms of chip area and power consumption are typically much higher than the achievable benefit in terms of speedup. For this reason, the *ParaNut* design uses ILP very conservatively and focuses on DLP and TLP instead.

Finally, the TLP category comprises multi-core architectures as well as all techniques for simultaneous multi-threading on a single core. From a hardware perspective, TLP requires a careful design of the memory/cache subsystem and bus interfaces. Besides this, a variable number of cores is a comfortable way to offer scalability (i.e. a performance-area trade-off) to the system integrator. From a software perspective, TLP requires the development of multi-threaded code, which is well understood today and supported by various software libraries and compiler extensions such as *OpenMP*.

To summarize, the design of the *ParaNut* architecture was guided by the following considerations:

- ILP techniques are used only very conservatively. The core architecture is optimized for area, not for speed.
- Good support for TLP and DLP should be provided to allow highly performant systems when needed.
- A programming model should be provided, that allows SIMD to be used easily using a high-level language and without the need for special libraries.

B. Instruction Set Architecture

The *ParaNut* instruction set architecture is compatible with the *OpenRISC 1000* specification. The *OpenRISC 1000* architecture is a 32-bit load and store RISC architecture designed with the purpose to support a spectrum of chip and system implementations [13]. Scalability is achieved by defining a minimalistic basic instruction set (*ORBIS32*) together with optional extensions including a floating-point unit (FPU) or a memory management unit (MMU). Furthermore, the basic architecture offers configuration options such as different register file sizes or optional arithmetic instructions.

ParaNut processors implement all mandatory instructions according to the *ORBIS32* specification. Hence, the *OpenRISC* GCC tool chain and libraries (newlib) can be used with the *ParaNut* without modifications. Features unique to *ParaNut*

require some additional *ParaNut*-specific instructions. These will be encapsulated in a small support library, so that they are still usable without compiler modifications.

C. Structural Organization

Figure 1 shows a block diagram of an exemplary *ParaNut* instantiation with four full-featured cores (alternatives will be discussed in Section IV). Each core contains an ALU, a register file and some control logic which together form the *Execution Unit (ExU)*. The instruction port (*IPort*) is responsible for fetching instructions from the memory subsystem and contains a small buffer for prefetching instructions. The data port (*DPort*) is responsible for performing the data memory accesses of load and store operations. It contains a small store buffer and implements write combining and store forwarding mechanisms as well as mechanisms to support atomic operations.

The *Execution Unit* is designed and optimized for a best-case throughput of one instruction in two clock cycles (CPI = 2, CPI = "clocks per instruction"). This is slower than modern pipeline designs targeting a best-case CPI value of 1. However, it allows to better optimize the execution unit for area, since no pipeline registers or extra components for the detection and resolution of pipeline conflicts are required. Furthermore, in a multi-core system, the performance is likely to be limited by bus and memory contention effects anyway, so that an *average* CPI value of 1 can hardly be achieved in practice. In the *ParaNut* design, several measures (such as the *IPort* and *DPort*) help to maintain an average-case throughput very close to the best-case value of CPI = 2, even for multi-core implementations.

The design of the memory interface and cache organization is very critical for the scalability of many-core systems. In a *ParaNut* system, the *Memory Unit (MemU)* contains the cache, the system bus interface, and a multitude of read and write ports for the processor cores. Each core is connected to the *MemU* by two independent read ports for instructions and data and one write port for data. The cache memory logically operates as a shared cache for all cores and is organized in independent banks with switchable paths from each bank to each read and write port. Tag data is replicated to allow arbitrary concurrent lookups. Parallel cache data accesses by different ports can be performed concurrently if their addresses a) map to different banks or b) map to the same memory word in the same bank. Furthermore, by using dual-ported Block-RAM cells, each bank can be equipped with two ports, so that up to two conflicting accesses (i. e. same bank, different addresses) are possible in parallel. Hence, even for many cores, the likelihood of contention can be arbitrarily reduced by increasing the number of banks, which is configurable at synthesis time.

The cache can be configured to be 1/2/4-way set associative with configurable replacement strategies (e.g. pseudo-random or least-recently used). The *Memory Unit* implements mechanisms for uncached memory accesses (e.g. for I/O ports) and support for atomic operations. All transactions to and from the system bus are handled by a bus interface unit, which presently supports the Wishbone bus standard, but can easily be replaced to support other busses such as AXI.

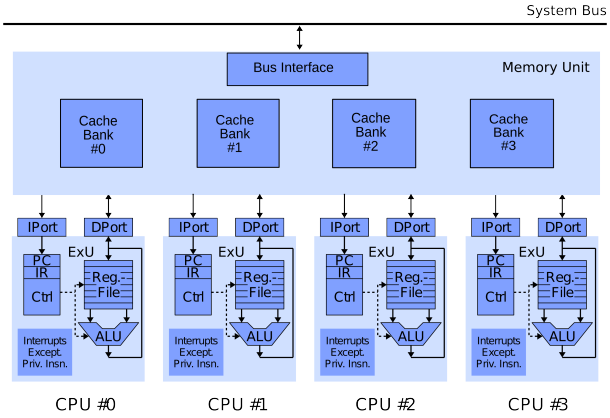

Fig. 1. Block diagram of a *ParaNut* with 4 (full-featured) cores

D. Software Development

For the processor system described so far, a cycle-accurate behavioral model has been written using SystemC. In the design phase, it has been used for performance analysis to support the design decisions described above. It serves both as a reference for the synthesizable VHDL implementation and as an instruction-level simulator.

For software development, the GCC tool chain from the OpenRISC project can be used. An operating environment based on the "newlib" C library allows to compile and run software both in the simulator and on real hardware (see Section VI).

IV. SIMD Using Standard Instructions

The SIMD extensions of popular embedded or workstation processors share some properties that make it difficult to develop optimizing compilers utilizing such units automatically [12]. These are, on the one hand, serious restrictions, such as fixed vector widths, a lack of conditional execution of individual vector elements and only limited memory addressing modes. On the other hand, common SIMD units contain very specialized instructions with complex semantics (e.g. "add with saturation"). They are very useful for performance-critical code sections commonly found in signal or image processing applications, for example. However, they can hardly be generated automatically by a compiler and are thus rarely seen in auto-vectorized code.

The *ParaNut* introduces a new concept for SIMD vectorization with a special focus on good compiler support for easy software development. While not fully implemented yet, this section describes the main ideas of this concept, which uses a multi-threading-like programming model for SIMD vectorization. It allows, on the one hand, to utilize SIMD vectorization using standard machine instructions, which, on the other hand, can easily be ported to other processors without SIMD extensions.

A. Linking Cores

Let us assume a normal 4-core processor as shown in Figure 1 with one modification: The control units of all but the first core are switched off, and instead, the control signals

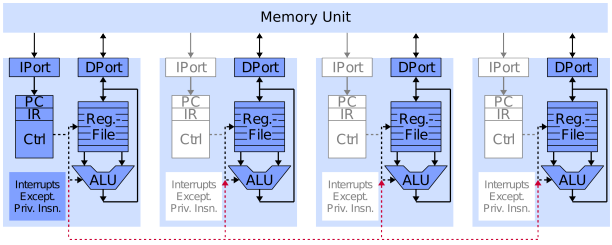

Fig. 2. *ParaNut* operating in *Linked Mode*

of the first core are also used by all other cores (see Figure 2). What would happen, if a multi-core processor is modified like this? From a hardware perspective, all instruction- and control-related components of the linked cores become superfluous. The remaining hardware strongly resembles a single-core CPU with an SIMD unit. There is just one instance of instruction fetch and decoding logic, but 4 parallel ALUs, register files and data ports, each of which can be seen as a vertical slice of a 128 (= 4 * 32) bit wide SIMD unit. The omitted hardware components save a considerable amount of logic area and - not less importantly - do not emit any memory accesses for instruction fetches. This strongly reduces the load on the cache subsystem and avoids system bus contention.

On the other hand, from a software perspective, the structure still resembles a multi-core architecture. To be precise, it still behaves *exactly* like a normal multi-core system, as long as no conditional branch instructions and jumps with variable target addresses occur in the instruction stream (jump and call instructions with fixed target addresses are still allowed). Hence, as long as these instructions do not occur, multiple parallel threads are actually executed by simple SIMD hardware. Or, from another point of view, SIMD vectorization can be programmed in the same way as multi-threaded code using the standard instruction set and the programmer's favorite language. There is no need for special instructions.

For example, the C code fragment

```
int n, a[4], b[4], w[4], wsum[4];
...
for (n = 0; n < 4; n++)
  wsum = a[n] * w[n] + b[n] * (100 - w[n]);
...
```

can be transformed as follows:

```
int n, a[4], b[4], w[4], wsum[4];
...
n = pn_begin_linked (4);
  // turns on linked mode, returns core id (0..3)
wsum = a[n] * w[n] + b[n] * (100 - w[n]);
pn_end_linked ();
  // switches back to single-thread mode
...
```

Initially, the processor runs in a single-thread mode with the primary core executing code and the linked cores being inactive. The macro *pn_begin_linked()* activates the linked cores with the effect that the subsequent code is synchronously executed 4 times in parallel. Conversely, the macro *pn_end_linked()* marks the end of the parallel section and causes the linked cores to be deactivated again.

Similarly to the two linked-mode macros, two macros *pn_begin_threaded(int nThreads)* and *pn_end_threaded()* can be defined which open and close a parallel section for conventional multi-threaded code. For (multi-core) processors that do not support the linked mode, the linked-mode macros can be mapped to their threaded-mode variants, and the code still executes correctly in multiple parallel threads. Hence, source code containing sections for the *ParaNut's* linked mode still remain fully portable.

As mentioned above, certain restrictions apply to code inside a linked-mode section. They are related to conditional branches and variable jump targets and will be discussed now. In C, the critical operations that require conditional branches are "if" statements, "case" statements and all loop statements. These constructs are not completely forbidden, but require special care. "if-else" statements can often be transformed such that both bodies are executed, and the final results are copied to the result variable(s) by conditional-move machine instructions. For example, a code fragment like

```
if (n < 0) {
  s = a - b;
}
else {
  s = a + b;
}
```

is not allowed in a parallel section, but the following code is:

```
s0 = a - b;
s1 = a + b;
pn_cmov (s, n < 0, s0, s1);
  // equivalent to "s = n < 0 ? s0 : s1;"
```

Loops are allowed with the premise that the repeat condition evaluates equally for all parallel threads. For example, the following for-loop is possible in linked mode since the loop condition only contains a variable which always has an identical value in all threads:

```
int a[1000], b[1000], s[1000];
int n, id;
...
// Sequential version...
for (n = 0; n < 1000; n += 1)
  s[n] = a[n] + b[n];
...
// Parallel version...
id = pn_begin_linked (4);
for (n = 0; n < 1000; n += 4)
  // Note: n is always identical in all threads
  s[n + id] = a[n + id] + b[n + id];
pn_end_linked ();
```

B. Modes and Capabilities

A *ParaNut* core can operate in one of 4 different modes, which are sketched in Figure 3. In mode 0, the core is inactive (i. e. presently not needed). In mode 1, the core operates in linked mode as explained above. A core running in mode 2 can execute a thread autonomously. It supports all standard instructions, but no interrupts, exceptions or system instructions. Finally, a processor in mode 3 represents a full-featured CPU. Often, only one core needs to support mode 3 allowing to save a considerable amount of complexity for the other cores while still maintaining all options for efficient thread-level parallelism.

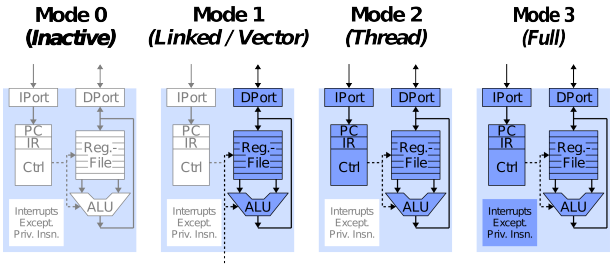

Fig. 3. Modes of a *ParaNut* core

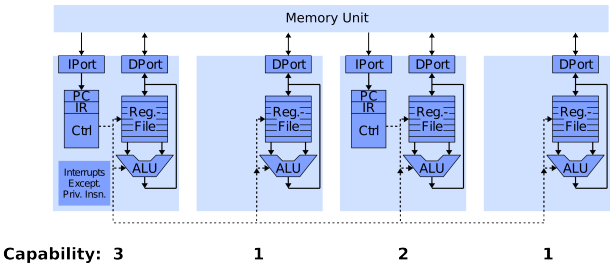

Fig. 4. Example of a *ParaNut* instantiation with cores of different capabilities

The available modes can be selected at synthesis time. The system designer decides about the number of cores and their *capability levels*, where a capability level of n means that the core can assume modes lower than or equal to n at runtime. Figure 4 shows a *ParaNut* instance with one capability-3, one capability-2 and two capability-1 cores. At runtime, this processor can arbitrarily be configured to execute, for example, two threads in parallel or one thread with 4-way SIMD-vectorized code (see Figure 5).

V. Implementation Status

In order to evaluate the *ParaNut* on FPGA hardware, a VHDL model has been implemented. Although having only been synthesized for a Xilinx FPGA in this paper, the model is designed to be platform independent by using inference for design primitives and therefore should be synthesizable for other FPGA devices as well. The overall VHDL implementation is still unoptimized in several places, leaving potential for improving clock frequency and resource usage in a future version. Also provided are features such as instruction or memory traces for simulation. Currently, only cores with the highest capability of 3 are supported, leaving potential for reducing logic usage in the future.

The VHDL model closely adheres to the architectural features of the SystemC model on which it is based. The *ExU* implements all required as well as most optional ORBIS32 instructions (except for multiply-accumulate, divide, find first and find last operations). Atomic operations are not yet implemented, but the primitives are already supported by the *MemU*. Exception handling has been implemented but remains to be verified. Integration of the *ParaNut* into a Wishbone based system-on-chip is supported by a Wishbone B4 compatible bus interface which provides burst accesses to the external memory bus.

Configurable buffer sizes for *IPorts/DPorts* as well as different implementations for bit shifting operations (serial

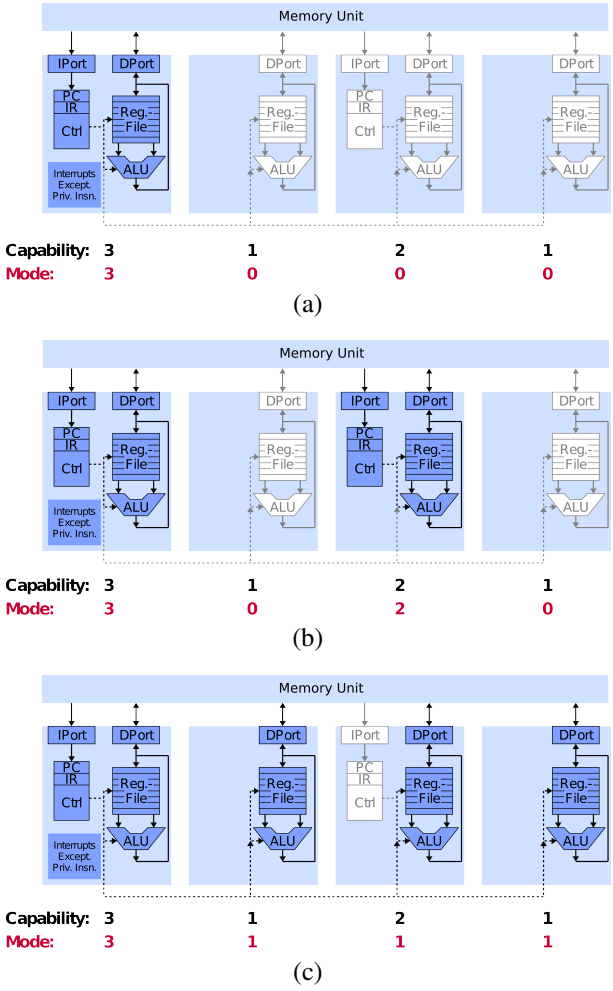

Fig. 5. *ParaNut* of Figure 4 running a) single-threaded code, b) two independent threads, and c) 4-way vectorized SIMD code

shifter, barrel shifter) and a configurable number of multiplier pipeline stages allow to trade speed for size and vice versa in the core itself.

The *MemU* allows for fine-grained control of cache parameters (cache associativity, number of sets, number of banks, replacement strategy, access arbitration). They can be chosen to best fit the task at hand for different numbers of CPU cores. For example, while the performance of an eight-core *ParaNut* can be increased by providing a cache layout with more banks to allow for simultaneous accesses of multiple cores, a single-core *ParaNut* will not be impeded by accesses from other cores and therefore does not suffer in (cache-)performance if only a single bank is implemented. Note, however, that the current VHDL implementation supports only a minimum of at least two banks. Block-RAM with two ports is inferred for every bank of cache memory, which allows for two read accesses or one write access of any *IPort* or *DPort* at a time. Since each bank RAM port needs to be routed to each *IPort* and *DPort*, the number of ports per bank can be reduced to one in order to save area at the expense of a slightly lower memory access performance.

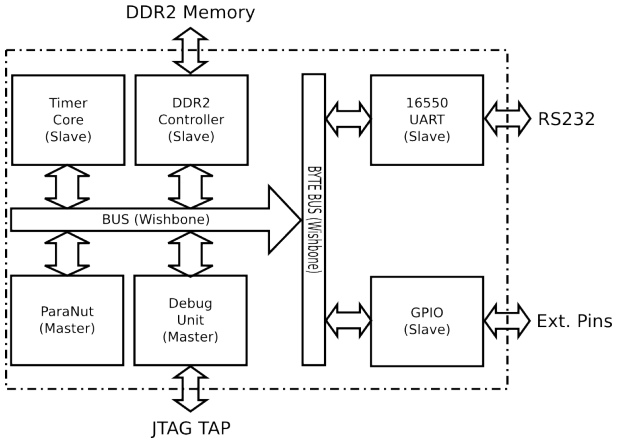

Fig. 6. Block diagram of the evaluation platform system-on-chip

VI. Experimental Results

A. Setup

A series of *ParaNut* variants have been synthesized and integrated into the *OpenRISC Reference Platform System-on-Chip (ORPSoCv2)* which can be used to build a Wishbone based system-on-chip together with an *OpenRISC 1200 (OR1200)* core. An existing implementation for the Xilinx ML501 board was ported to the Xilinx ML509 board (XUPV5-LX110T) featuring a Virtex-5 LX110T FPGA. Using the same platform, the *OR1200* could then be directly compared to the *ParaNut* by simply replacing the *OR1200* CPU with the *ParaNut* in the FPGA design. Figure 6 shows the *ParaNut* and additional Wishbone-based peripheral cores that support the evaluation by measuring time and printing output to the UART. The *OpenRISC USB-JTAG debugger* was connected to the *Debug Unit*, and together with the *OpenRISC Debug Proxy* application they provide an interface for GDB which allowed to transfer program data into DDR2 memory.

For evaluating performance in benchmarks the *ParaNut* cores were configured to achieve high performance at a low area consumption. The *IPorts* and *DPorts* were configured with 4 word buffers. The *ExU* included a barrel shifter and a 3-stage pipelined multiplier. A cache size of 256 KB was chosen to allow all benchmark programs to completely reside in the cache. This avoids slow main memory accesses and puts more emphasis on the performance of the cache system and the processor itself during measurements. The cache is 4-way set-associative with a least-recently used replacement strategy. The specific cache configuration of each *ParaNut* implementation is detailed in Table I.

TABLE I. Configuration of cache parameters for different ParaNut implementations.

Cores	Banks	BRAM ports	Size
1	2	1	256KB
2	2	2	256KB
4	4	2	256KB
8	4	2	256KB

For the *OR1200* core, the instruction and data caches were both configured to be 32 KB in size, which was the maximum supported setting. However, benchmark results showed no significant decline with only 16 KB of instruction and data cache.

B. Synthesis Results

The *ParaNut* variants and an *OR1200* system have been synthesized using the *Xilinx ISE 14.7* software suite. Table II shows slice usage of the *ParaNut* with 1 to 8 cores and other open-source soft core processors. The column "Cores" denotes the number of physical (*ParaNut, OR1200, LEON3*) or logical cores (*OpenSPARC T1*), respectively.

TABLE II. Virtex-5 slice usage and maximum frequency for ParaNut and OR1200

Processor	Cores	LUT/FF pairs	MHz
ParaNut	1	3,387	75
	2	6,803	73
	4	14,168	48
	8	29,691	32
OR1200	1	3,286	76
LEON3 [4]	1	3,500 (LUT only)	125
OpenSPARC T1 [14], [15]	1	31,475 (LUT only)	50
	4	51,558 (LUT only)	10

The *ParaNut* figures show, that the area increases almost linearly with the number of cores. This is expected due to the replication of logic for the cores (each comprising an *ExU*, *IPort* and *DPort*) and the complexity of the *MemU* which is growing with the addition of more banks and ports. Presently, the maximum clock rate decreases with an increasing number of cores. This is due to the combinational arbitration logic inside the *MemU*, which is subject to ongoing optimization. For the future, the goal is to achieve a clock rate independent from the number of cores in the order of 70 MHz for this technology.

The slice usage of the single-core *ParaNut* is in a similar order of magnitude for the *OR1200* or the *LEON3* processor, even though the *ParaNut* supports a complex 4-way set-associative cache of 256 KB. On the other hand, area of a 1-thread single-core *OpenSPARC T1* implementation on the same Virtex-5 FPGA is nearly 10-fold with the 4-thread core being the maximum to fit on the Virtex-5 device. Note, that the entries for LUT/FF pairs for *LEON3* and *OpenSPARC T1* show the number of lookup tables only, and that the total slice usage is typically higher.

C. Benchmark Results

The performance of the *ParaNut* has been evaluated and compared to other soft-core processors using the *Dhrystone* and *CoreMark* benchmarks. The benchmarks were run with a core clock frequency of 25 MHz. All benchmark programs were compiled with the GCC version 4.5.1-or32-1.0rc4 utilizing the "newlib" C library. Compiler optimization level was set to 3 ("-o3") and compiler flags corresponding to CPU capabilities were set according to the hardware configurarion ("-mhard-mul -msoft-div -msoft-float").

Tables III and IV show the results for the *Dhrystone* and *CoreMark* benchmarks, respectively. *Dhrystone* is not multi-threaded and uses integer division, so only single-core variants without hardware divider have been considered. *CoreMark* was executed in multi-threaded mode. The column "Speedup" in Table IV shows the speedup relative to the single-core *ParaNut*.

TABLE III. DHRYSTONE BENCHMARK RESULTS

Processor	DMIPS/MHz	Speedup
ParaNut	0.288	1.0
OR1200	0.345	1.2

TABLE IV. COREMARK BENCHMARK RESULTS

Processor	Cores	CoreMark/MHz	Speedup
ParaNut	1	0.80	1.0
	2	1.60	2.0
	4	3.17	4.0
	8	6.13	7.7
OR1200	1	1.28	1.6
LEON3 [4]	1	1.8	2.3
MicroBlaze [16]	1	1.9	2.4
Nios II [16]	1	0.93 .. 1.60	1.2 .. 2

The results are promising: Certainly, the performance of a single-core *ParaNut* still lags somewhat behind that of the other processors reported in Table IV, which are all in a mature development state and have been optimized for many years now. However, the *ParaNut* architecture has been designed for a high level of parallelism, and these efforts appear to be successful. Although a shared memory model and a single, shared cache for all cores is used, even 8 cores achieve a speedup of 7.7, and for fewer cores, the performance increases almost ideally with the number of cores. The *LEON3* officially only supports a maximum of 4 cores and the *OR1200* is not multi-core capable at all.

VII. CONCLUSION AND FUTURE PROSPECTS

The *ParaNut* architecture is a new open and highly scalable processor architecture for embedded systems. Special focus is put on parallelism at thread and data level in order to allow small or powerful systems based on one architecture. A new concept of parallel processing units with customizable capabilities will allow a seamless transition between SIMD vectorization and thread-level parallelism. The core architecture has been implemented and evaluated on an FPGA. The benchmark results appear very promising, especially the speedup of 7.7 with 8 cores indicates an excellent scalability.

The project is still in an early stage, and the present implementation leaves a lot of room for optimizations and improvements. Future work will concentrate on completing the implementation and evaluating the advantages of the presented linked mode. A support library is planned to provide a POSIX-Threads-compatible interface, so that the SIMD and threading capabilities can be utilized using OpenMP. Further steps will be to develop an MMU and to support Linux as an operating system.

REFERENCES

[1] D. G. Bailey, *Design for Embedded Image Processing on FPGAs*, 1st ed. Wiley Publishing, 2011.

[2] M. Pohl, M. Schaeferling, and G. Kiefer, "An efficient FPGA-based hardware framework for natural feature extraction and related Computer Vision tasks," in *24th International Conference on Field Programmable Logic and Applications (FPL)*, Sept. 2014, pp. 1–8.

[3] M. Schaeferling, M. Bihler, M. Pohl, and G. Kiefer, "ASTERICS - An Open Toolbox for Sophisticated FPGA-Based Image Processing," in *13th embedded world Conference*, Feb. 2015.

[4] Aeroflex Microelectronic Solutions, "LEON3 Processor," 2014. [Online]. Available: http://www.gaisler.com/index.php/products/processors/leon3

[5] Adapteva Inc., "Epiphany Multicore Intellectual Property," 2015. [Online]. Available: www.adapteva.com/epiphany-multicore-intellectual-property/

[6] P. Mishra and N. Dutt, *Processor Description Languages*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008.

[7] P. Ienne and R. Leupers, *Customizable Embedded Processors: Design Technologies and Applications*, ser. Series in Systems on Silicon. Massachusetts: Morgan Kaufmann, Jul. 2006.

[8] R. Inta, D. J. Bowman, and S. M. Scott, "The Chimera: An Off-the-shelf CPU/GPGPU/FPGA Hybrid Computing Platform," *International Journal of Reconfigurable Computing*, vol. 2012, pp. 2:2–2:2, Jan. 2012. [Online]. Available: http://dx.doi.org/10.1155/2012/241439

[9] S. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. Taylor, and R. Laufer, "PipeRench: a coprocessor for streaming multimedia acceleration," in *Proceedings of the 26th International Symposium on Computer Architecture*, 1999, pp. 28–39.

[10] M. Lanuzza, S. Perri, P. Corsonello, and M. Margala, "A New Reconfigurable Coarse-Grain Architecture for Multimedia Applications," in *Second NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, Aug. 2007, pp. 119–126.

[11] J. Choi, S. Brown, and J. Anderson, "From software threads to parallel hardware in high-level synthesis for FPGAs," in *International Conference on Field-Programmable Technology (FPT)*, Dec. 2013, pp. 270–277.

[12] J. L. Hennessy and D. A. Patterson, *Computer Architecture - A Quantitative Approach (5. ed.)*. Morgan Kaufmann, 2012.

[13] D. L. et al., *OpenRISC 1000 Architecture Manual*. opencores.org, April 2014. [Online]. Available: http://opencores.org/or1k/Main_Page

[14] Oracle Corporation, "OpenSPARC Slide-Cast," 2008. [Online]. Available: http://www.oracle.com/technetwork/systems/opensparc/2008-oct-opensparc-slide-cast-07-tt-1539014.html

[15] Oracle Corporation, "OpenSPARC T1 on Xilinx FPGAs – Updates," 2008. [Online]. Available: http://www.oracle.com/technetwork/systems/opensparc/18-ramp-2008-final-1530382.pdf

[16] Embedded Microprocessor Benchmark Consortium (EEMBC), "CoreMark Benchmark Scores Database," 2015. [Online]. Available: http://www.eembc.org/coremark/index.php